Font Subsetting and Downloading in the PostScript Printer Driver of Qt/X11

Sivan Toledo
School of Computer Science, Tel-Aviv University
Tel-Aviv 69978, Israel

Lars Knoll
Trolltech AS
Waldemar Thranes gate 98, N-0175 Oslo, Norway

Abstract

This paper describes the font discovery, subsetting, and downloading mechanism in Qt/X11. The mechanism addresses a major usability issue: prior to the implementation of this mechanism, users of Qt applications (and hence users of KDE) could not print non-Latin text, and could only print Latin text in fonts that are built into most printers. The new mechanism allows users to print text in any script that Qt/X11 supports, which includes western scripts (primarily Latin, Cyrillic, and Greek), Arabic, Hebrew, and east-Asian scripts. The new mechanism also allows users to print Latin text using almost any PostScript Type 1 or TrueType font that X11 supports. The mechanism usually finds font files without any configuration beyond that required to use the fonts under X11.

1 Introduction

Qt/X11, the X toolkit that KDE uses, includes a printer driver that allows applications to render text and graphics on a PostScript device. Prior to Qt version 2.3.0, this printer driver had very limited WYSIWYG text-rendering capabilities. It could only render text if it could guess correctly the PostScript font name from the XLFD name, and if the font was resident at the printer. Essentially, the driver only supported a limited range of standard Latin fonts and one symbol font. This meant, for example, that users could use Konqueror, KDE's web browser, to view Hebrew web pages, but could not print them. Users could also not use nonstandard Latin fonts in otherwise sophisticated applications, such as KWord and KPresenter, KDE's word processor and presentation programs. This situation, which is still quite common in X applications, is clearly below the current level of users' expectations.

Versions 2.3.0 and later of Qt/X11 include a font discovery, subsetting, and downloading mechanism that rectifies this problem. The mechanism enables true WYSIWYG text rendering in print jobs for all Qt (and therefore KDE) applications running under X, as long as the X fonts that they use are generated from TrueType or Type1 fonts. Figure 1 demonstrates this capability.

The font downloading and subsetting mechanism in the driver

1. Finds the font file corresponding to a given X screen font, and
2. uses the font file to insert a scalable description of the glyphs that are used in the document into the PostScript output. This operation is referred to as subsetting the font and downloading it into the print file.

We had three main design goals in mind when we designed and implemented this mechanism:

1. Support for the most common scalable font formats: TrueType and PostScript Type 1.
2. Support for Unicode, which Qt uses internally for almost all strings.
3. The ability to print with any TrueType or PostScript Type 1 font that X11 uses without any configuration files beyond those required to simply use the fonts under X11.

As the paper shows, we have essentially achieved our goals. While the first two goals are self explanatory, the

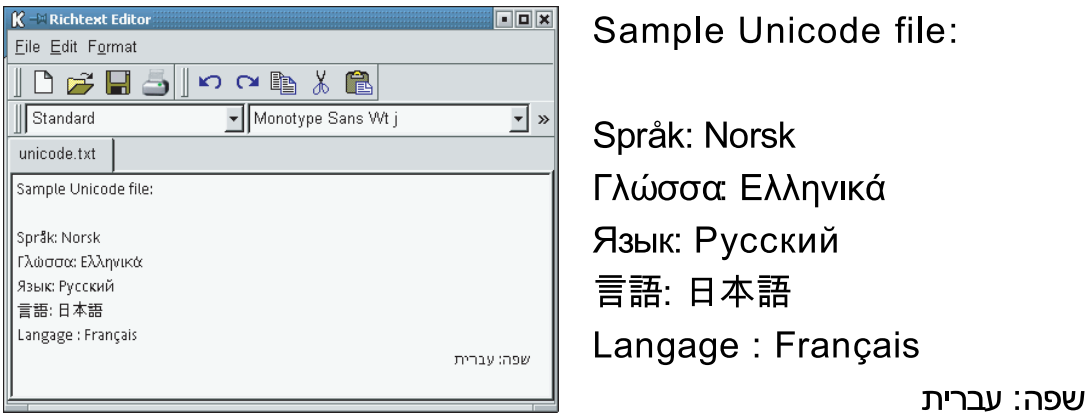

Figure 1: A simple Qt example program running under X11 (left) and its printed output (right). The printed output on the left shows only part of the printed page.

third requires some explanation. A mechanism that requires additional configuration files fails when they are missing or defective. A mechanism that requires no configuration is, therefore, much more robust. We wanted to maximize the chances that a font that the user sees on the screen is downloaded correctly to a print job, and the lack of configuration files helps us achieve this goal.

The rest of the paper is organized as follows. Section 2 presents background on fonts and on text rendering in PostScript. Section 3 describes the overall structure of Qt/X11's printer driver, to which we have added the new mechanism. The mechanism itself is described in Section 4. Section 5 suggest additional features that would benefit users if added to the font-handling mechanism that we describe. Section 6 explains why the existence of multiple printer drivers in the X world, each with its own font-handling mechanism, harms users. The section suggests that these mechanisms be unified into a single font-subsetting-and-downloading mechanism. Section 7 summarized the paper.

2 Background

2.1 Font Files

Digital fonts allow programs to render text on output devices such as monitors and printers. A digital font consists of three main components, which can reside in a single or in multiple files. The first component consists of *glyph descriptions*, which describe the shape of letters, parts of letters, or groups of letters. The second component of a font is an *encoding* or a set of encodings. An encoding maps the characters of a character set, such as ASCII or Unicode, to glyphs. The third component consists of *metrics* and other layout information, which assists the application in laying out text. Fonts also contain auxiliary information, such as the name and style of the font, copyright information, and so on.

Most fonts today contain *scalable* glyph descriptions using either cubic or quadratic splines. A software component called *a rasterizer* uses these curves to decide which pixels are covered by the glyph and should be painted and which should be left unpainted. So-called *antialiased* rasterizers paint pixels in several colors to simulate the effect of partially-covered pixels. Glyph descriptions usually usually contain data called *hints* in addition to the splines. Hints help the rasterizer draw better-looking glyphs at low resolutions. *Bitmap* glyph descriptions that specify explicitly which pixels to paint, and which were once common, are becoming rarer. Some font formats allow *composite* glyphs, which represent a single character using appropriately placed base glyphs. For example, the glyph for the character 'a with a grave accent' can be represented by translated references to the glyph representing 'a' and to the glyph representing the accent.

An encoding maps the characters of a character set to glyphs in the font. Glyphs are specified using indices or using symbolic names. Some font formats avoid the use of encodings by putting the glyphs in an array whose size is the length of the encoding, so glyph indices directly correspond to character codes. But most font formats today include explicit encodings. The encoding is used by the rasterizer, which uses it to draw the correct glyph for each character. The application sometimes uses the encoding as well in order to access metric and layout

information associated with specific glyphs.

All fonts contain at least one kind of glyph-specific metrics—the width of each glyph. The width of the glyph allows text-layout applications to measure text in order to compute line breaks and to justify text, and it allows the renderer to determine where to draw the next glyph. Many fonts contain other metrics, such as metrics for vertical text layout, for pair kerning (bringing specific pairs of glyphs closer together or further apart) and so on. Some fonts also contain additional text-layout data, such as ligature substitution information (replacing consecutive glyphs by a single glyph that represents multiple letters, such as a glyph representing an 'f' followed by an 'i', as in file), glyph positioning information (for attaching multiple diacritics to a single letter, for example), and glyph substitution information (when multiple glyphs are available for representing a single letter).

We now explain the main features of the most widely-used scalable font formats in use today.

TrueType Fonts. TrueType fonts [8] store all the font information in a single data file (usually with suffix `ttf`) containing tables (data structures) that contain glyph descriptions, encodings, and so on. The glyphs are described using quadratic splines and composition of base glyphs. Glyphs are referred to using indices, although many TrueType fonts also specify names for the glyphs. TrueType fonts are hinted using programs in a special programming language. These programs can move the control points of the splines to fit better a low resolution pixel grid. The metrics of the font can also be modified using these programs, so a 10-point font may have different glyph widths at 100 PPI (pixels per inch) than at 1200 PPI. Most TrueType fonts contain a Unicode encoding, and some contain additional encodings. *TrueType collection* fonts (extension

PostScript Type 1 Fonts. Type 1 fonts [1, 6], which will be described in more details below, store font data in two data files. One data file, with suffix `pfb` or `pfa`, stores the encoding, glyph descriptions (in either binary or ASCII, hence the two suffixes), and the width of glyphs. Another file contains additional metric and layout information. The auxiliary file may be binary (a `pfm` file) or ASCII (a `afm` file). Glyphs are referred to using symbolic names, and they are described using cubic splines or using translated composition. Hints in Type 1 fonts are declarative, which means that the font designer declares explicitly certain important features of the glyph, but it is up to the rasterizer to decide how to use this information. For example, a hint may declare that the two counters (empty spaces) in the letter 'm' should be exactly equal in width. Type 1 fonts are 8-bit fonts, which means that they use an encoding that maps characters in the range 0–255 to named glyphs. Most Type 1 fonts contain unencoded glyphs, which means that no character maps to them. Programs can reencode the font, or replace its encoding, which allows them to access all the glyphs in the font.

OpenType Fonts. There are two types of OpenType fonts [9]. The first type is a TrueType font with additional tables, mostly tables that contain advanced layout information. Such fonts are also valid TrueType fonts so the can be processed using any program that processes TrueType fonts. Therefore, from here on we refer to OpenType fonts with TrueType outlines as TrueType fonts.The second type is similar to TrueType fonts except that the glyph descriptions and the hints use a representation similar to that of Type 1 fonts. These fonts, which have an `otf` format, allow for lossless conversion of Type 1 fonts to 'almost' TrueType format. OpenType fonts are relatively new, but quite a few fonts are available in this format from Adobe. Many of these OpenType fonts contain advanced typographic features, such as old-style figures, small capitals, glyph variants, and so on.

2.2 PostScript Text Rendering

PostScript documents are programs in the PostScript language that contain rendering commands. A Font is described in PostScript using a *dictionary*, the main data structure in the PostScript language.

Four main PostScript commands (with some variants) render text. The `findfont` command retrieves an already defined font. The `scalefont` command scales a font to a given point size. The `setfont` assigns a previously found and scaled font to be the fonts in which subsequent text will be rendered. The `show` command draws a string of text in the current font.

QT/X11's printer driver actually uses a variant of the `show` command called `ashow`. This command has three arguments, d_x, d_y, and a string. The command renders the string but adds d_x to the width of each character and d_y to its "vertical width". This allows the driver to fit a string into a given length. The driver uses this ability

to ensure that text on paper takes exactly the same horizontal space as the same text on the screen, even though the length on the screen is constrained to whole pixels.

2.3 PostScript Fonts

Thirteen fonts (Times, Helvetica and Courier in 4 styles and Symbol) are built into all PostScript devices. Most PostScript devices have additional built-in fonts. But to render text in a font that is not built into the device, the PostScript document must include a representation of the font. The process of including a representation of the font in the PostScript file is called *font downloading*.

PostScript interpreters support several types of downloadable fonts.

Type 1 Fonts. PostScript document may include Type 1 fonts in ASCII representation. The PostScript document includes only the glyph descriptions and the encoding, not the additional metric and layout information. All PostScript devices support Type 1 fonts.

Type 3 Fonts. These fonts [6] are the most general glyph representation of any PostScript fonts. To rasterize a glyph in a Type 3 font, the PostScript interpreter invokes a procedure that the font provides. The procedure is invoked with two arguments, the font dictionary and the character to be drawn (or the name of the glyph in Level-2 and -3 PostScript). The procedure can use all the tools of the PostScript language to draw the glyph. It can invoke other procedures, draw bitmap images, and draw shapes defined by cubic splines. It can also determine the pixel grid that is being painted and adapt the glyph to the pixel grid. Because Type 3 fonts can utilize all the power of the PostScript language, font rasterizers that do not include a PostScript interpreter cannot process them. In particular, unlike Type 1 fonts, which can be used on Windows, Mac, and X11 systems, Type 3 fonts cannot be used in these windowing environments. They are primarily used to download non-Type-1 fonts into PostScript documents. For example, `Dvips` generates Type 3 fonts to download bitmapped fonts into PostScript documents, and some Windows printer drivers generate Type 3 fonts to download TrueType fonts into PostScript documents. Like Type 1 fonts, Type 3 fonts also use an 8-bit encoding vector.

Type 42 Fonts. Type 42 fonts [5, 6] are PostScript wrappers for TrueType fonts. A Type 42 font dictionary contains PostScript string (or several strings) that encode the original TrueType font, a data structure that maps glyph indices to symbolic names, and an 8-bit encoding vector that maps characters to named glyphs. Some level-1 and -2 PostScript devices cannot rasterize Type 42 fonts; All level-3 devices and many level-2 ones can; GhostScript can. There are two variants of Type 42 fonts: with or without a GlyphDirectory. Using a GlyphDirectory is more flexible but not all devices that support Type 42 fonts supports GlyphDirectories.

Type 0 Fonts. Type 0 fonts [6] are composite fonts that are designed for supporting character sets with more than 256 characters. Type 0 fonts contain no glyph descriptions or metrics. They only contain a mapping from characters to glyphs in one or more 8-bit base fonts. For example, a Type 0 font may map Unicode characters to glyphs from several Type 1 fonts. Composite fonts support several mapping mechanisms. The mapping mechanism that we use is called an 8/8 mapping. Each character to be rendered is represented using two bytes, where the first byte selects the base font and the second selects the glyph using the base font's 8-bit encoding vector. Other mapping mechanisms allow for other splittings of 8-bit and 16-bit characters, as well as for statefull character encodings and for CID mappings, described below.

CID Fonts. CID fonts [3, 2, 6] are special Type 0 fonts that are designed to allow a single encoding to map characters to the glyphs of many fonts. A CID font contains glyph descriptions for all the glyphs of a specified *character collection*. A *character map* (CMap) maps the characters of a character set to the glyphs of a character collection. To use a CID font, the PostScript program composes a CID font with a character map to form a CID-Keyed font. The CID font and the character map must, obviously, use the same character collection. This arrangement allows a single encoding mechanism (the character map) to be used with many different fonts, as long as all the fonts contain the same glyph set in the same order. For example, composing the font `Munhwa-Regular` with the character map `UniKS-UCS2-H` creates the font `Munhwa-Regular--UniKS-UCS2-H` ,which has a 16-bit Unicode encoding. CID fonts can have Type 1 glyph descriptions, Type 3 glyph descriptions, bitmap glyph descriptions, or TrueType glyph descriptions. CID fonts are used today solely for CJK (Chinese, Japanese, Korean) fonts (see [7]).

2.4 Subsetting and Incremental Definition of Fonts in PostScript

The representation of a font in a PostScript document needs not include all the glyphs in the original digital font files. The printer driver can include in the file only the description of glyphs that are actually used in a document. *Subsetting* a font requires that the driver that downloads the font be able to manipulate the data structures of the font file. In particular, whenever a font includes composite glyphs, including a composite in a subset requires that all the base glyphs of the subset be included as well.

Some fonts are so large that subsetting them is virtually a necessity. TrueType fonts such as Times New Roman and Courier New, which include Latin, Greek, Cyrillic, Hebrew, and Arabic glyphs, have over 1300 glyphs and their files are over 300KBytes in size. Fonts such as Bitstream Cyberbit, Arial Unicode MS, and IBM Times New Roman WorldType, which include CJK glyphs, have tens of thousands of glyphs and their file are 13–24MBytes in size. The representation of these fonts as Type 42 PostScript fonts would be even larger. Such fonts must be subsetted.

PostScript fonts can be defined incrementally. That is, the first definition of a font main describe only a proper subset of the glyphs that the document uses. When additional glyphs are needed later in the document, they are added to the font dictionary. The addition of glyphs obeys the PostScript scoping rules of `save/restore` blocks. Therefore, when the block in which the glyphs are added ends, their descriptions are removed from the font.

Incremental font definition serves two purposes. First, the fact that glyphs are removed when the `save/restore` block ends allows the driver to produce a PostScript document that can be processed with less memory. Since pages are typically enclosed between a `save` and a `restore`, the driver can build a document in which each page adds the glyph descriptions required for that page. In such cases the PostScript interpreter never needs to store a large set of glyphs from a font. This technique benefits mainly CJK fonts, in which the set of glyphs used in a long document may be considerably larger than the set used on any particular page. The negative implications of this techniques are a larger PostScript file size and longer processing, since the same glyph may need to be processes on several pages.

The second use for incremental definition of fonts is when the driver must begin producing the PostScript output before it finishes processing the document. In such cases, it must define the fonts that are used before it can determine the exact subset of their glyphs that are used in the document. The driver can then either download the entire glyph set of the fonts, or it can define fonts with the glyph set that has been used up to that point, and add additional glyphs as it encounters them in the document.

Glyphs can be added to Type 1 and 3 fonts, and to Type 42 fonts with a GlyphDirectory.

3 The Qt/X11 Printer Driver

The Qt/X11 printer driver is the software module in Qt that enables applications to generate PostScript output for printing, storage, or further processing (e.g., conversion to PDF).

To generate PostScript output, the application creates a QPrinter object, which is a QPaintDevice by inheritance. The QPrinter object specifies the name of the printer (or the file to write the output into), the paper dimensions, and so on. Typically, the application creates the QPrinter object by creating a QPrintDialog, which lets the user choose the printer and the printer setup and. The application then asks the QPrintDialog to create a QPrinter object. Once the QPrinter object is ready, the application uses it to create a QPainter. QPainter is an object that provides a drawing API to the application. The application sends drawing commands to the QPainter, which sends them to the QPrinter. This part of the mechanism works in the same way on all the platforms that Qt supports, including both X11 and Windows.

Under X11, the QPrinter object contains a reference to a QPSPrinter object. The QPSPrinter object, in turn, contains a reference to a QPSPrinterPrivate object, which performs the actual PostScript generation. The reasons for the existence of the intermediate QPSPrinter object are irrelevant to this paper.

When the application draws on the QPainter associated with the QPrinter, the QPainter sends the drawing requests to its associated QPrinter (which the application constructed, usually using a QPrintDialog). The QPrinter sends the drawing commands to its QPSPrinter, which sends them to its QPSPrinterPrivate object, which generates the appropriate PostScript. The QPSPrinter-

Private object represents the state of the PostScript print driver for a single print job. The methods of the QPSPrinterPrivate class comprise the printer driver itself.

Two commands that the application passes to the printer driver are related to text rendering: setfont, which sets the font and size in which text is rendered from that point on, and drawtext, which actually renders text. The setfont command specifies the font by passing to the driver a QFont object, which is essentially a Qt object representing an X11 screen font. On systems running XFree86 4 and later, the X11 screen font can be either a core X font or an Xft font [10].

The printer driver (QPSPrinterPrivate) attempts to buffer the PostScript document that it generates. A PostScript document usually consists of a header, which includes fonts and other resources that the document needs, and of page descriptions. Buffering the document allows the driver to construct a header that includes all the needed resources. To keep the buffer size reasonable, the driver emits the contents of the buffer into the output stream when the PostScript output grows beyond certain limits. Before it emits the contents of the buffer, the driver generates and emits a header that includes the resources that the document needs up to that point. Since the driver has not yet accepted drawing commands for the rest of the document, it does not know which additional fonts and resources the following pages need. From that point on, the driver emits one page at a time. It needs to include with a page description any resources that the page needs and that were not included in the document's header.

4 The New Mechanism

The new font subsetting and downloading mechanism is built around a new class, QPSPrinterFontPrivate. This class is used solely by the printer driver and its interface is not in Qt's public API. QPSPrinterFontPrivate is a base class that represents a PostScript font corresponding to a Qt screen font. The subclasses of the base class correspond to specific font formats: QPSPrinterFontTTF for TrueType fonts, QPSPrinterFontPFA and QPSPrinterFontPFB for Type 1 fonts, QPSPrinterFontAsian for built-in CJK fonts, and QPSPrinterFontNotFound for screen fonts for which no matching font file was found. There are four subclasses of QPSPrinterFontAsian, corresponding for Japanese, Korean, traditional-Chinese and simplified-Chinese fonts. Some of the new subsetting and downloading functionality is implemented in the QPSPrinterFontPrivate base class and the rest is implemented in its subclasses.

4.1 Finding Font Files

The setfont command passes a Qt screen-font object to the driver. The driver keeps a reference to the screen font and returns.

The drawtext command asks the driver to render a string in the current font. The driver checks whether the last PostScript font that was used matches the current screen font (the argument of the last setfont command). If not, it calls its own `setFont` method to change the current PostScript font. Either way, the driver now has a reference to the QPSPrinterPrivate object that corresponds to the current screen font. It then calls this object's `drawText` method to render the string.

The driver's `setFont` method creates a temporary object of type QPSPrinterFont. It passes to QPSPrinterFont's constructor references to the screen font, to the driver object itself, and to the script (language) that the screen font implements. The script is only used to select a built-in CJK font if no font file is found. The QPSPrinterFont constructor performs three tasks. First, it extracts a canonical name for the screen font. For core X fonts, the name consists of the first 5 fields in the font's XLFD name, which include the foundry (vendor), the family name, the weight (e.g., bold), the slant (e.g., italic), and the width; it ignores the size fields and the encoding fields in the font's name, since PostScript fonts can be resized and reencoded. If the screen font is an Xft font [10], the canonical name is simply the font's file name, which Xft provides. The next task is to determine whether a font with the same canonical name has already been used in the document. This is determined by searching a dictionary data structure that maps the canonical names of the document's fonts to QPSPrinterFontPrivate references. If the font has been used, processing ends here. If the font has not been used, the third task of the QPSPrinterFont's constructor is to find and read the font file. For Xft fonts, this task is trivial. For core X fonts, the task is more complex. The driver searches the `fonts.dir` and `fonts.scale` files on the X font path and on the font server's font path for matching XLFD names. The X font path can be determined exactly by calling `XGetFontPath`. The X font server's font path cannot be determined using the API of either X or the font server itself. The code therefore guesses locations for the font server's configuration file and tries to parse this file in order to determine the font server's font path. This heuristic succeeds on standard

configurations but is likely to fail with nonstandard font servers, such as Bitstream's Fontastic, which is part of Corel's Linux applications. The user or system administrator can also add directories to be searched by setting the `/qt/fontPath` application setting.

Once QPSPrinterFont finds the font file, it reads it into a buffer (if it was not read before) and determines the font format. The first few bytes of the font file can determine unambiguously the font format: binary Type 1 fonts have `0x80` in their first byte and the string `%!PS` starting in byte 6, ASCII Type 1 fonts start with `%!PS` , and TrueType fonts start with the 4-byte string `0x00010000`, and so on. Now that QPSPrinterFont knows which format the font is in, it constructs an appropriate instance of a subclass of QPSPrinterFontPrivate. The constructor is given a reference to the buffer containing the font file. The last action of QPSPrinterFont is to insert the newly created QPSPrinterFontPrivate into the document's font dictionary. The QPSPrinterFont is not used any more and is destroyed.

The only other important action of the driver's `setFont` method is the generation of PostScript code to set the current font. This is done by calling the QPSPrinterFontPrivate, which emits the required PostScript code into the PostScript buffer.

4.2 Drawing Text

When the driver is requested to draw a string, it calls the `drawText` method of the current QPSPrinterFontPrivate. This method performs two actions. First, it measures the width of the string emits into the PostScript buffer an `ashow` command that renders the string. The width measurement is done using the screen-font's metrics, to ensure that printed output matches the appearance of text on the screen.

Second, the method adds the Unicode characters in the string (all Qt's strings are encoded in Unicode) to the subset of characters that the downloaded font must support.

The string to be rendered must be encoded in the PostScript document using the PostScript font's encoding. That is, the driver must map Unicode characters to the font encoding. Font encodings in the driver have evolved considerably; the mechanism that we now describe applies to Qt 3.0, but not to Qt 2.x.

Qt 3.0 always uses a 16-bit encoding (except for built-in Japanese fonts, where it uses the jisx0208.1983-0 encoding). The encoding of a font is constructed incrementally. The first character in the encoding is always the default `.notdef` character. The first character that is rendered using the font in the document is assigned slot 1 in the encoding, the second slot 3, and so on. Therefore, each font typically has a different encoding depending on the characters that are rendered in it and on their order of appearance.

An earlier version of the driver used the Unicode encoding for all the fonts, but this usually led to larger PostScript output without any significant benefit.

4.3 Subsetting and Downloading Fonts

The current driver downloads Type 1 and TrueType fonts, but it only subsets TrueType fonts.

Type 1 fonts are downloaded entirely; the current implementation does not subset them. This is not usually a significant problem, since of the Latin Type 1 fonts are relatively small, 50-100KBytes in the ASCII encoding that the driver downloads into the PostScript output. Binary Type 1 fonts are converted to ASCII and downloaded, ASCII Type 1 fonts are downloaded without any changes. Subsetting Type 1 fonts would make the PostScript output smaller but it requires parsing the glyph descriptions, which the current driver does not do.

TrueType fonts are converted into PostScript fonts and sometimes subsetted. Most of the code that performs the conversion was originally written by David Chappell for a project called PPR. The code converts the TrueType font either to a Type 42 font, which is simply a TrueType font in a PostScript wrapper, or to a Type 3 font that uses PostScript-language procedure to describe the outline of each glyph.

By default, TrueType fonts are converted into Type 3 fonts. The default distribution of Qt/X11 does not even compile the Type 42 conversion. The required code can be included in the library by defining a certain preprocessor variable. Even if the code is included, the driver generates Type 42 fonts only if the `QT_TTFTOPS` environment variable is set to 42.

The driver prefers conversion to Type 3 for two main reasons. First, the driver only subsets Type 3 fonts, not Type 42 fonts. Subsetting Type 3 fonts is relatively easy: the driver simply skips the glyph descriptions of glyphs that are not used. The driven ensures, however, that if

a character in the subset is represented by a composite glyph, then all the required base glyphs are included in the subset font. Subsetting a Type 42 font is more complex and we have not implemented the required code. Since the driver does not subset Type 42 fonts, conversion to Type 42 produces larger output files. The conversion actually fails on very large TrueType fonts, since in a Type 42 font, all the TrueType tables except for the table containing the glyph descriptions (the `glyf` table) must fit into PostScript strings whose maximum size is 64KBytes. Large fonts can have tables that are too large. Second, not all PostScript devices support Type 42 fonts. Since the driver does not know whether the PostScript output file will be rendered on a device that can rasterize Type 42 fonts, it prefers to rely on Type 3 fonts which are supported by all PostScript devices.

Conversion of TrueType fonts to Type 42 has one potential advantage over conversion to Type 3. In conversion to Type 42 the TrueType font is essentially embedded as is, including its hints. Our conversion to Type 3 represents the glyph outlines exactly, but strips the hints. At small sizes, striping the hints can have an adverse effect on the printed output. However, a few subjective visual tests that we have performed showed little difference in printed output even for well-hinted fonts like Arial. It seems that at printer resolutions, as opposed to screen resolutions, hinting has little effect.

To support fonts with more than 256 characters, the driver uses Type 0 fonts. The PostScript font that the driver downloads, whether Type 1, 3, or 42, uses a built-in 8-bit encoding vector, `StandardEncoding`, but the driver does not use this encoding; it is included in the font only to make it a valid font. Suppose that the document uses 300 characters from a font, say Georgia. The base PostScript font, which has an 8-bit encoding, is called `Georgia`. The driver creates two 8-bit encoding vectors, one containing the first 255 characters that it needs from the font (and `.notdef`), the other containing the other 45 characters. The first encoding is called `Georgia-ENC-00` and the second `Georgia-ENC-01`. The driver then creates two derived 8-bit fonts in which the Georgia is reencoded using the two encoding vectors. These derived fonts are called `Georgia-Uni-00` and `Georgia-Uni-01`. Finally, the driver defines a composite Type 0 font with an 8/8 mapping that maps characters 0–255 to `Georgia-Uni-00` and characters 256–511 to `Georgia-Uni-01`.

Using Unicode to encode the composite fonts is also possible, and indeed an earlier version of the driver used such an encoding.

5 Suggestions for Future Development of the Printer Driver

The driver can benefit from several additional features that we have not yet implemented.

Conversion of TrueType Fonts to Type 1. The driver currently converts TrueType fonts to Type 3 or Type 42 fonts. Conversion to Type 1 instead of Type 3 would reduce output sizes and would speed up rasterization. Type 1 use a special encoding for outline glyph descriptions that is more compact than the Type 3 fonts that the driver produces now. Furthermore, the compact encoding allows for faster processing by the PostScript interpreter than the ASCII-encoded Type 3 fonts (Adobe even produced at one time a hardware accelerator for rasterizing Type 1 fonts [7, page 287]). What is needed to implement this feature is a Type 1 encoder; several open-source programs include such an encoder (e.g., `t1asm` from the `t1utils` package), so the code can probably be borrowed from one of them.

Subsetting of Type 1 Fonts. The driver currently downloads Type 1 fonts without subsetting them. Subsetting Type 1 fonts would reduce somewhat the output file sizes that the driver produces. Subsetting Type 1 fonts requires that the driver decodes the glyph descriptions, which it does not currently do. It would then need to encode the description of the glyphs that the document needs. Again, similar code is available in `t1asm` and `t1disasm`. Subsetting requires care, since the subset font must obey all the constraints on Type 1 fonts, such as constraints on composite glyphs and on subroutines. Some programs, such as `dvips`, TEX's PostScript backend, subsets Type 1 fonts, but not always correctly. Since Type 1 fonts are typically small, it is obviously better not to subset them than to introduce bugs into the driver by attempting to subset.

Tailoring the Output to Specific PostScript Devices. PostScript devices differ in their capabilities. Devices differ in the amount of memory they have, in the selection of built-in-fonts that they have, in the PostScript language features that they support, and in the area of the page that they can print on, and so on. Therefore, a PostScript document that is optimal for one device may render poorly on another, or it may not render at all. The current driver ignores the specifics of the output device and always attempts to generate "generic" PostScript

that renders on any device. This strategy has several significant drawbacks: it results in large file sizes, especially when the document contains hi-resolution images, it prevents the driver from converting TrueType fonts to Type 42 fonts, and it prevents the driver from notifying applications about the imageable area of the page.

There are two approaches to tailoring PostScript output to specific devices. In one approach, a dialog widget allows the user to tailor the output. For example, the PostScript output dialog of Adobe Illustrator 9.0 for Windows allows the user to select between PostScript level-1, level-2, or level-3, to decide whether to download fonts into the document, and to decide on the color model. Other high-end applications offer similar capabilities, and so does Adobe Acrobat for Linux. In the other approach, the driver uses a *PostScript Printer Description (PPD)* file to determine the capabilities of the output device. The driver uses the PPD files in two ways. First, it can determine automatically whether some device features are available and adapt the output accordingly, without any input from the user. For example, the driver can determine, using the PPD file, that the device is a PostScript level-2 device with support for Type 42 fonts. Second, the driver can determine which capabilities of the printer are under software control and ask the user to decide how to process the document. For example, the driver can determine that printer X can print on both sides of the paper (duplex capability), so it incorporates in the dialog widget a checkbox for duplex printing, whereas printer Y can only print on one side, so this checkbox is not included in the dialog widget. Adobe's generic PostScript printer drivers for Windows and MacOS use this approach.

The first approach is useful when the PostScript output is intended to be used on several devices or to be further processed. For instance, when users produce PostScript to be put on a web site, or when users produce an encapsulated PostScript (EPS) figure to be included in other documents. The first approach is also useful when the printer driver cannot determine the PPD file that corresponds to a particular output device. Most Unix/Linux printing systems do not associate PPD files with printers, so the PPD-less approach is more suitable to them. The main disadvantages of the first approach, compared to the second, is that it presents the users with options that they may not understand, and that it makes it nearly impossible to support all the features that a PostScript device may have. A user may not know the difference between PostScript levels or what it means to download fonts, so including such options in dialogs can be confusing. Also, some PostScript devices have fairly exotic capabilities, such as stapling and binding, and it is unlikely that a generic and easy-to-use interface would support all of them.

It seems that the most reasonable solution for this driver would be to allow the user to control the output without using PPD files. To avoid confusion, we think that the extra options should not be in the main printing dialog, but accessible through an "advanced options" button or a similar mechanism. This makes it clear that setting the extra options is not necessary and that it requires some expertise.

Support for Additional Font Formats. The vast majority of scalable fonts that are used on X11 systems today are Type 1 and TrueType fonts, so by virtue of supporting them the driver supports most of the fonts that users have. But FreeType, the font rasterization library that is used by the XFree86 X server, by Xft, and by some stand-alone font servers, can rasterize fonts in other formats as well (`www.freetype.org`). The most important font formats that FreeType supports besides Type 1 and TrueType are OpenType fonts and CID fonts with Type 1 outlines. Since FreeType supports these formats, XFree86 and Xft can support them. If such fonts become widely used on X11 systems, then the driver should support them. OpenType fonts are currently only available from Adobe, and they are not widely used. On the other hand, Adobe announced that it plans to convert its entire font library, which is currently offered in Type 1 format, to OpenType format, and Windows 2000 supports OpenType, so such fonts may become more widely used in the future.

Regarding CID fonts, these are used solely for CJK fonts; we do not know whether support for CID fonts is important to users of X11 systems.

Adding support for bitmap fonts, and in particular, to the X core fonts, would create a more robust fall-back rendering mechanism when the driver cannot find the font file or cannot download it. In such cases, the driver can simply retrieve the bitmap glyphs using the X protocol (or using Xft if the font is an Xft font) and download a PostScript font with bitmap glyph descriptions. The resulting output may look coarse, since it would contain scaled-up versions of low-resolution bitmaps, but it would at least render all the glyphs that the user can see on the screen.

6 Why Users Need a Unified Font-Handling Mechanism for Printer Drivers

Quite a few other X11 libraries and programs include a PostScript printer driver that can download fonts. These include StarOffice (through a commercial printer-driver called Xprinter, `http://www.bristol.com/xprinter`), AbiWord, Sun's JDK and JRE, Wine (and in particular, the version of Wine distributed with Corel's Linux applications). The overall functionality in all of these drivers is basically similar: they provide a drawing API to the application and produce PostScript output. The drawing API's of the various drivers differ, but not by much. The capabilities of the drivers, in terms of font processing, are similar but not uniform. For example, Xprinter only supports Type 1 fonts, but it supports PPD files, which the other drivers do not.

The existence of several different drivers harms users, for two reasons. First, not all the fonts that the user has work in all applications and in all situations. For example, a TrueType font that works fine in Corel's PhotoPaint does not work in StarOffice. Or a commercial font that works fine in some applications does not work in StarOffice because StarOffice depends on the font metric file (`.afm`), and the AFM parser sometimes fails on valid files. Some situations are even more confusing to users, as when a font works in an application in one locale but not in another, even though the font's glyph repertoire supports both locales. The second reason that multiple font-handling drivers harm users is that fonts are hard to install. To use a new font, the user typically needs to configure each application (that is, each driver). StarOffice has a font installation dialog, and so do Corel's applications. AbiWord only uses fonts that are stored in (or linked to) its own font directory, so to use a new font with AbiWord, the user must copy the font to (or create a link in) AbiWord's font directory and update configuration files in that directory. This situation also means that when new applications are installed, they typically only use the fonts that came with them, not fonts that are already installed on the system.

The best way to fix the problem is to unify the printer drivers, or at least the font-handling component of the drivers. A unified font-handling mechanism would mean that a a font only needs to be installed once, and that once a font works in one application, it works in all applications. And as the font subsetting and downloading mechanism of Qt/X11 shows, printer drivers can support fonts without any configuration besides that required for X11 itself.

Tools like `kfontinst` take a different approach: they provide the user with a unified font-installation interface, but they attempt to configure multiple driver to use newly-installed fonts. This solution is an effective stop-gap measure, but we think that in the long run unifying the mechanisms is preferable. Tools like `kfontinst` need to be configured themselves, in order to find all the required configuration files; they typically do not support all drivers and applications (e.g., `kfontinst` only configures X11 and StarOffice); specific applications may still fail to handle fonts that other applications handle well. A unified font-handling mechanism suffers from none of these problems.

The simplest way to unify the font-handling mechanisms of printer drivers is using a stand-alone library. The API of QPSPrinterFontPrivate can serve as a first draft to the API of this library. Another option is to add the required features to FreeType, since significant amounts of code in FreeType can be reused for font subsetting and downloading. Another good reason to add this functionality to FreeType is that environments that need to rasterize fonts often also need to download fonts to print jobs. However, the font-file discovery mechanism that we use does not seem appropriate for FreeType since it is X11 specific. Another option is to include the font-handling mechanism in a generic printer-driver library, which could perhaps support not only PostScript, but other page-description languages as well, such as HP's PCL. This is obviously a larger-scale project.

7 Summary

This paper describes the font discovery, subsetting, and downloading mechanism in Qt/X11. The mechanism addresses a major usability issue: prior to the implementation of this mechanism, users of Qt applications (and hence users of KDE) could not print non-Latin text, and could only print Latin text in fonts that are built into most printers. The new mechanism allows users to print text in any script that Qt/X11 supports, which includes western scripts (primarily Latin, Cyrillic, and Greek), Arabic, Hebrew, and east-Asian scripts. Thai and Indic scripts probably need additional support to render properly. The new mechanism also allows users to print Latin text using a wide variety of fonts.

The mechanism achieves our main design goals, which were support for common font formats, support for Uni-

code, and lack of configuration files.

The paper presents possible enhancements to the printer driver and suggests that a unified font-subsetting-and-downloading mechanism would be beneficial to both developers and users.

References

[1] Adobe Systems. *Adobe Type 1 Font Format*. Available online on `http://partners.adobe.com/asn/developer/technotes/main.html`. 1990.

[2] Adobe Systems. *CID-Keyed Font Technology Overview.* Adobe Technical Note #5092, available online on `http://partners.adobe.com/asn/developer/technotes/main.html`. 1994.

[3] Adobe Systems. *Adobe CMap and CID Font Files Specification, Version 1.0.* Adobe Technical Note #5014, available online on `http://partners.adobe.com/asn/developer/technotes/main.html`. 1996.

[4] Adobe Systems. *PostScript Printer Description File Format Specification, Version 4.3*. Adobe Technical Note #5003, available online on `http://partners.adobe.com/asn/developer/technotes/main.html`. February 1996.

[5] Adobe Systems. *The Type 42 Font Format Specification.* Adobe Technical Note #5012, available online on `http://partners.adobe.com/asn/developer/technotes/main.html`. 1998.

[6] Adobe Systems. *PostScriptő Language Reference Manual*. Third edition. Available online on `http://partners.adobe.com/asn/developer/technotes/main.html`. 1999.

[7] Ken Lunde. *CJKV Information Processing*. O'Reilley, 1999.

[8] Microsoft. *TrueType 1.0 Font Files, Technical Specification Revision 1.66.* Available online on `http://www.microsoft.com/typography`, August 1995.

[9] Microsoft. *OpenType Specifications Version 1.3*. Available online on `http://www.microsoft.com/typography`, April 2001.

[10] Keith Packard. Design and implementation of the X rendering extension. Proceedings of *Usenix 2001*, FREENIX track. 12 Pages. Boston, June 2001.